

# Fully Dynamic All Pairs All Shortest Paths\*

Matteo Pontecorvi and Vijaya Ramachandran

University of Texas at Austin, USA  
 {cavia,vlr}@cs.utexas.edu

## Abstract

We consider the all pairs all shortest paths (APASP) problem, which maintains all of the multiple shortest paths for every vertex pair in a directed graph  $G = (V, E)$  with a positive real weight on each edge. We present a fully dynamic algorithm for this problem in which an update supports either weight increases or weight decreases on a subset of edges incident to a vertex. Our algorithm runs in amortized  $O(\nu^{*2} \cdot \log^3 n)$  time per update, where  $n = |V|$ , and  $\nu^*$  bounds the number of edges that lie on shortest paths through any single vertex. Our APASP algorithm leads to the same amortized bound for the fully dynamic computation of betweenness centrality (BC), which is a parameter widely used in the analysis of large complex networks.

Our method is a generalization and a variant of the fully dynamic algorithm of Demetrescu and Italiano [7] for unique shortest paths, and it builds on very recent work on decremental APASP [19]. Our algorithm matches the fully dynamic amortized bound in [7] for graphs with unique shortest paths, though our method, and especially its analysis, are different.

## 1 Introduction

Given a directed graph  $G = (V, E)$  with positive edge weights, we consider the problem of maintaining *APASP* (*all pairs all shortest paths*) [19], i.e., the set of all shortest paths between all pairs of vertices. This is a fundamental graph property, and it also enables the efficient computation of *betweenness centrality* (BC) for every vertex in the graph. BC is a widely used measure for the analysis of social networks.

In this paper we present a fully dynamic algorithm for the APASP problem, where each update in  $G$  is either *incremental* or *decremental*. An incremental update either inserts a new vertex along with incident edges of finite weight, or decreases the weights of some existing edges incident on a vertex. A decremental update deletes an existing vertex, or increases the weights of some edges incident on a vertex.<sup>1</sup>

Recently, we gave (with Nasre) a simple incremental APASP algorithm [18], and a more involved decremental APASP algorithm [19]. Neither of these algorithms is correct for the fully dynamic case. The fully dynamic method that we present in this paper builds on [19], and is a variant of the fully dynamic method developed by Demetrescu and Italiano [7] for APSP (the ‘DI’ method) where only one shortest path is maintained for each pair of vertices. The DI algorithm runs in  $O(n^2 \cdot \log^3 n)$  amortized time per update, where  $n = |V|$ . A method that is faster by a logarithmic factor was given by Thorup [27], but this algorithm is considerably more complicated, even for the unique shortest paths case. For instance, the experimental study in [8] considers the DI method and a few other methods, but omits the algorithm in [27], deeming it to be mainly of theoretical interest. The algorithms in both [7] and [27] use the unique shortest paths assumption crucially, and need considerable enhancements even to maintain a small number of multiple shortest paths

\*This work was supported in part by NSF grants CCF-0830737 and CCF-1320675. Authors’ affiliation: Dept. of Computer Science, University of Texas, Austin, TX 78712. An extended abstract of the results in this paper is in *Proc. ISAAC* 2015.

<sup>1</sup>Incremental/decremental refer to the insertion/deletion of a vertex or edge; the corresponding weight changes that apply are weight decreases/increases, respectively.

correctly. In this paper we present a fully dynamic algorithm for APASP, which maintains all of the multiple shortest paths for every pair of vertices. Our algorithm is as simple as the DI method (though somewhat different) when specialized to unique shortest paths, and it matches the DI bound for graphs with a constant number of (or unique) shortest paths while being applicable to the more general APASP problem. Our APASP algorithm gives a fully dynamic algorithm for the betweenness centrality (BC) problem, described below.

**Betweenness Centrality (BC).** Betweenness centrality is a widely-used measure in the analysis of large complex networks, and is defined as follows. For any pair  $x, y$  in  $V$ , let  $\sigma_{xy}$  denote the number of shortest paths from  $x$  to  $y$  in  $G$ , and let  $\sigma_{xy}(v)$  denote the number of shortest paths from  $x$  to  $y$  in  $G$  that pass through  $v$ . Then,  $BC(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$ . This measure is often used as an index that determines the relative importance of  $v$  in the network. Some applications of BC include analyzing social interaction networks [13], identifying lethality in biological networks [20], and identifying key actors in terrorist networks [5, 15]. Heuristics for dynamic betweenness centrality with good experimental performance are given in [10, 16, 25], but none of these heuristics provably improve on the widely used static algorithm by Brandes [4], which runs in  $O(mn + n^2 \log n)$  time (where  $m = |E|$ ) on any class of graphs. The only earlier results that provably improve on Brandes on some classes of graphs are the recent separate incremental and decremental algorithms in [18, 19]. Table 1 contains a summary of these results.

It is shown in [18, 19] that a decremental APASP algorithm can be used to give a decremental BC algorithm with the same bound. In a similar way, our fully dynamic APASP algorithm gives a fully dynamic BC algorithm with the same bound. We do not give further details for fully dynamic BC here.

Paper	Year	Time	Weights	Update Type	DR/UN	Result
Brandes [4]	2001	$O(mn)$	NO	Static Alg.	Both	Exact
Brandes [4]	2001	$O(mn + n^2 \log n)$	YES	Static Alg.	Both	Exact
Geisberger et al. [9]	2007	Heuristic	YES	Static Alg.	Both	Approx.
Riondato et al. [23]	2014	depends on $\epsilon$	YES	Static Alg.	Both	$\epsilon$ -Approx.
<b>Semi Dynamic</b>						
Green et al. [10]	2012	$O(mn)$	NO	Edge Inc.	Both	Exact
Kas et al. [12]	2013	Heuristic	YES	Edge Inc.	Both	Exact
NPR [18]	2014	$O(\nu^* \cdot n)$	YES	Vertex Inc.	Both	Exact
NPRdec [19]	2014	$O(\nu^{*2} \cdot \log n)$	YES	Vertex Dec.	Both	Exact
Bergamini et al. [3]	2015	depends on $\epsilon$	YES	Batch (edges) Inc.	Both	$\epsilon$ -Approx.
<b>Fully Dynamic</b>						
Lee et al. [16]	2012	Heuristic	NO	Edge Update	UN	Exact
Singh et al. [25]	2013	Heuristic	NO	Vertex Update	UN	Exact
Kourtellis+ [14]	2014	$O(mn)$	NO	Edge Update	Both	Exact
Bergamini et al. [2]	2015	depends on $\epsilon$	YES	Batch (edges)	UN	$\epsilon$ -Approx.
<b>This paper</b>	2015	$O(\nu^{*2} \cdot \log^3 n)$	YES	Vertex Update	Both	Exact

Table 1: Related results (DR stands for Directed and UN for Undirected)

**Our Results.** Let  $\nu^*$  be the maximum number of edges that lie on shortest paths through any given vertex in  $G$ . Our main result is the following theorem, where we assume for convenience  $\nu^* = \Omega(n)$ .

**Theorem 1.** *Let  $\Sigma$  be a sequence of  $\Omega(n)$  fully dynamic APASP updates on an  $n$ -node graph  $G = (V, E)$ . Then, APASP and all BC scores can be maintained in amortized time  $O(\nu^{*2} \cdot \log^3 n)$  per update, where  $\nu^*$  bounds the number of distinct edges that lie on shortest paths through any given vertex in any of the updated graphs or their vertex induced subgraphs.*

**Discussion of the Parameter  $\nu^*$ .** Let  $m^*$  be the number of edges in  $G$  that lie on shortest paths. As noted in [11], it is well-known that  $m^* = O(n \log n)$  with high probability in a complete graph where edge weights are chosen from a large class of probability distributions. Since  $\nu^* \leq m^*$ , our algorithms will have an amortized bound of  $O(n^2 \cdot \text{polylog}(n))$  on such graphs. Also,  $\nu^* = O(n)$  in any graph with only a constant number of shortest paths between every pair of vertices. These graphs are called  $k$ -geodetic [22] (when at most  $k$  SPs exists between two nodes), and are well studied in graph theory [26, 1, 17]. In fact  $\nu^* = O(n)$  even in some graphs that have an exponential number of SPs between some pairs of vertices. In contrast,

$m^*$  can be  $\Theta(n^2)$  even in some graphs with unique SPs, for example the complete unweighted graph  $K_n$ . Another type of graph with  $\nu^* \ll m^*$  is one with large clusters of nodes (e.g., as described by the *planted  $\ell$ -partition model* [6, 24]). Consider a graph  $H$  with  $k$  clusters of size  $n/k$  (for some constant  $k \geq 1$ ) with  $\delta < w(e) \leq 2\delta$ , for some constant  $\delta > 0$ , for each edge  $e$  in a cluster; between the clusters is a sparse interconnect. Then  $m^* = \Omega(n^2)$  but  $\nu^* = O(n)$ .

In all such cases our algorithm will run in amortized  $O(n^2 \log^3 n)$  time per update. Thus we have:

**Theorem 2.** *Let  $\Sigma$  be a sequence of  $\Omega(n)$  updates on graphs with  $O(n)$  distinct edges on shortest paths through any single vertex in any vertex-induced subgraph. Then, APASP and all BC scores can be maintained in amortized time  $O(n^2 \cdot \log^3 n)$  per update.*

**Corollary 3.** *If the number of shortest paths through any single vertex is bounded by a constant, then fully dynamic APASP and BC have amortized cost  $O(n^2 \cdot \log^3 n)$  per update if the update sequence has length  $\Omega(n)$ .*

Our algorithm uses  $\tilde{O}(m \cdot \nu^*)$  space, extending the  $\tilde{O}(mn)$  result in DI for APSP. Brandes uses only linear space, but all known dynamic algorithms require at least  $\Omega(n^2)$  space.

In a companion paper [21], we present a fully dynamic APASP algorithm that gives a logarithmic factor improvement over the results in Theorems 1 and 2, and Corollary 3. This algorithm is considerably more complex than the algorithm we present in the current paper.

**Overview of the Paper.** In Section 2 we review the DI fully dynamic algorithm as well as the decremental APASP algorithm in [19]. In contrast to the DI method, the decremental algorithm in [19] is not a correct fully dynamic APASP algorithm. In Section 3 we describe a basic fully dynamic APASP algorithm, which includes suitable enhancements to [19] to be correct for fully dynamic updates. However, this algorithm is not very efficient. In Section 4 we present FULLY-DYNAMIC, our overall fully dynamic algorithm, which is similar to the DI fully dynamic algorithm but uses a different ‘dummy update’ sequence. In Section 5 we analyze our algorithm and establish the amortized time bound claimed in Theorem 1. The paper ends with a conclusion in Section 6.

## 2 Background

Our fully dynamic APASP algorithm builds on the approach used in the elegant fully dynamic APSP algorithm of Demetrescu and Italiano [7] for unique shortest paths (the ‘DI’ method). It also builds on the decremental APASP algorithm in [19] (the ‘NPR’ method). We now briefly review these two methods, and then we give an overview of our method for fully dynamic APASP. The rest of the paper presents our method and its amortized complexity.

### 2.1 The NPR Decremental APASP Algorithm [19]

The decremental algorithm NPR for APASP builds on the key concept of a *locally shortest path (LSP)* in a graph, introduced in the DI method [7]. A path  $p$  in  $G$  is an LSP if the path  $p'$  obtained by removing the first edge from  $p$  and the path  $p''$  obtained by removing the last edge from  $p$  are both SPs in  $G$ . For APASP, we need to maintain all shortest paths, and  $G$  can have an exponential (in  $n$ ) number of SPs. Thus the DI method is not feasible for APASP since it maintains each SP (and LSP) separately. In order to succinctly maintain all SPs and LSPs in a manner suitable for efficient decremental updates, NPR developed the tuple system described below.

#### 2.1.1 A System of Tuples [19]

Since a graph could have an exponential number of shortest paths, NPR introduced the compact tuple system described below (we briefly review it, referring the reader to [19] for more details). Let  $d(x, y)$  denote the shortest path length from  $x$  to  $y$ .

A *tuple*,  $\tau = (xa, by)$ , represents a set of paths in  $G$ , all with the same weight, and all of which use the same first edge  $(x, a)$  and the same last edge  $(b, y)$ . If the paths in  $\tau$  are LSPs, then  $\tau$  is an LST (locally shortest tuple), and the weight of every path in  $\tau$  is  $\mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$ . In addition, if  $d(x, y) = \mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$ , then  $\tau$  is a *shortest path tuple* (ST).

A *triple*  $\gamma = (\tau, wt, count)$ , represents the tuple  $\tau = (xa, by)$  that contains *count* number of paths from  $x$  to  $y$ , each with weight *wt*. We use triples to succinctly store all LSPs and SPs for each vertex pair in  $G$ . For  $x, y \in V$ , we define:

$$\begin{aligned} P(x, y) &= \{((xa, by), wt, count) : (xa, by) \text{ is an LST from } x \text{ to } y \text{ in } G\} \\ P^*(x, y) &= \{((xa, by), wt, count) : (xa, by) \text{ is an ST from } x \text{ to } y \text{ in } G\}. \end{aligned}$$

**Left Tuple and Right Tuple.** A left tuple (or  $\ell$ -tuple),  $\tau_\ell = (xa, y)$ , represents the set of LSPs from  $x$  to  $y$ , all of which use the same first edge  $(x, a)$ . A right tuple ( $r$ -tuple)  $\tau_r = (x, by)$  is defined analogously. For a shortest path  $r$ -tuple  $\tau_r = (x, by)$ ,  $L(\tau_r)$  is the set of vertices which can be used as pre-extensions to create LSTs in  $G$ , and for a shortest path  $\ell$ -tuple  $\tau_\ell = (xa, y)$ ,  $R(\tau_\ell)$  is the set of vertices which can be used as post-extensions to create LSTs in  $G$ . Hence:

$$\begin{aligned} L(x, by) &= \{x' : (x', x) \in E(G) \text{ and } (x'x, by) \text{ is an LST in } G\} \\ R(xa, y) &= \{y' : (y, y') \in E(G) \text{ and } (xa, yy') \text{ is an LST in } G\}. \end{aligned}$$

For  $x, y \in V$ ,  $L^*(x, y)$  denotes the set of vertices which can be used as pre-extensions to create shortest path tuples in  $G$ ;  $R^*(x, y)$  is defined symmetrically:

$$\begin{aligned} L^*(x, y) &= \{x' : (x', x) \in E(G) \text{ and } (x'x, y) \text{ is a } \ell\text{-tuple representing SPs in } G\} \\ R^*(x, y) &= \{y' : (y, y') \in E(G) \text{ and } (x, yy') \text{ is an } r\text{-tuple representing SPs in } G\}. \end{aligned}$$

**Data Structures.** The algorithm in [19] uses priority queues for  $P$ ,  $P^*$ , and balanced search trees for  $L^*$ ,  $L$ ,  $R^*$  and  $R$ , as well as for a set Marked-Tuples that is specific only to one update. It also uses priority queues  $H_c$  and  $H_f$  for the cleanup and fixup procedures, respectively. Some key differences between this representation and the one in DI [7] are described in [19].

**Lemma 4.** [19] *Let  $G = (V, E)$  be a directed graph with positive edge weights. The number of LSTs (or triples) that contain a vertex  $v$  in  $G$  is  $O(\nu^{*2})$ , and the total number of LSTs (or triples) in  $G$  is bounded by  $O(m^* \cdot \nu^*)$ .*

The NPR algorithm maintains all STs and LSTs in the current graph, and for each tuple, it maintains the  $L$ ,  $R$ ,  $L^*$  and  $R^*$  sets. To execute a new update to a vertex  $v$ , NPR (similar to DI) first calls an algorithm cleanup on  $v$  which removes all STs and LSTs that contain  $v$ . This is followed by a call to algorithm fixup on  $v$  which computes all STs and LSTs in the updated graph that are not already present in the system. The overall algorithm update consists of cleanup followed by fixup. If the updates are all decremental then NPR maintains exactly all the SPs and LSPs in the graph in  $O(\nu^{*2} \cdot \log n)$  amortized time per update. Several challenges to adapting the techniques in the DI decremental method to the tuple system are addressed in [19]. The analysis of the amortized time bound is also more involved since with multiple shortest paths it is possible for the dynamic APASP algorithm to examine a tuple and merely change its count; in such a case, the DI proof method of charging the cost of the examination to the new path added to or removed from the system does not apply.

## 2.2 The DI Fully Dynamic APSP Algorithm [7]

The DI method first gives a decremental APSP algorithm, and shows that this is also a correct, though inefficient, fully dynamic APSP algorithm. The inefficiency arises because under incremental updates the method may maintain some old SPs and their combinations that are not currently SPs or LSPs; such

paths are called historical shortest paths (HPs) and locally historical paths (LHPs). To obtain an efficient fully dynamic algorithm, the DI method introduces ‘dummy updates’ into the update sequence. A dummy update performs cleanup and fixup on a vertex that was updated in the past. Using a strategically chosen sequence of dummy updates, it is established in [7] that the resulting APSP algorithm runs in amortized time  $O(n^2 \cdot \log^3 n)$  per real update. The DI method continues to use the notation  $P^*$ ,  $L^*$ , etc., even though these are supersets of the defined sets in a fully dynamic setting. We will do the same in our fully dynamic algorithms.

## 2.3 Our Approach

A natural approach to obtain a fully dynamic APASP algorithm would be to convert the NPR decremental APASP algorithm to an efficient fully dynamic APASP algorithm by using dummy updates, similar to DI. There are two steps in this process, and each has challenges (the second step is more challenging).

**Step 1:** *Converting NPR decremental algorithm to a correct (but inefficient) fully dynamic APASP algorithm.* In Section 3 we describe algorithm FULLY-UPDATE, an updated version of the NPR decremental algorithm in [19], which is a correct fully dynamic APASP algorithm that maintains a superset of all STs and LSTs in the current graph. This algorithm, however, is not very efficient since it may add, remove, and examine a large number of tuples (this is similar to the DI decremental algorithm when used as a fully dynamic algorithm without any additional features).

**Step 2:** *Obtaining a good dummy sequence for efficient fully dynamic APASP.* In Section 4 we present Algorithm FULLY-DYNAMIC, which calls FULLY-UPDATE not only on the current update, but also on a sequence of vertices updated at suitable previous time steps. This is similar to the dummy updates used in DI, however we use a different dummy update sequence. We now describe the DI dummy update sequence and its limitations for APASP. Then, we introduce our new method (whose full details are in Section 4).

The DI method uses ‘dummy updates’, where a vertex updated at time  $t$  is also given a ‘dummy’ update at steps  $t + 2^i$ , for each  $i > 0$  (this update is performed along with the real update at step  $t + 2^i$ ). The effect of a dummy update on a vertex  $v$  is to remove any HP or LHP that contains  $v$ , thereby streamlining the collection of paths maintained. Further, with unique SPs, each HP in  $P^*(x, y)$  for a given pair  $x, y$  will have a different weight. An  $O(\log n)$  bound on the number of HPs in a  $P^*(x, y)$  is established in DI as follows. Let the current time step be  $t$ , and consider an HP  $\tau$  last updated at  $t' < t$ . Let us denote the smallest  $i$  such that  $t' + 2^i > t$  as the dummy-index for  $\tau$ . By observing that different HPs for  $x, y$  must have different dummy-indices, it follows that their number is  $O(\log t)$ , which is  $O(\log n)$  since the data structure is reconstructed after  $O(n)$  updates.

If we try to apply the DI dummy sequence to APASP, we are faced with the issue that a new ST for  $x, y$  (with the same weight) could be created at each update in a long sequence of successive updates. Then, an incremental update could transform all of these STs into HTs. If this happens, then several HTs for  $x, y$ , all with the same weight, could have the same dummy-index (in DI only one HP can be present for this entire collection due to unique SPs). Thus, the DI approach of obtaining an  $O(\log n)$  bound for the number of HPs for each vertex pair does not work for HTs in our tuple system.

Our method for Step 2 is to use a different dummy sequence, and a completely different analysis that obtains an  $O(\log n)$  bound for the number of different ‘PDGs’ (a PDG is a type of derived graph defined in Section 4) that can contain the HTs. Our new dummy sequence is inspired by the ‘level graph’ method introduced in Thorup [27] to improve the amortized bound for fully dynamic APSP to  $O(n^2 \cdot \log^2 n)$ , saving a log factor over DI. The Thorup method is complex because it maintains  $O(\log n)$  levels of data structures for suitable ‘level graphs’. Our algorithm FULLY-DYNAMIC does not maintain these level graphs. Instead, FULLY-DYNAMIC performs exactly like the fully dynamic algorithm in DI, except that it uses this alternate dummy update sequence, and it calls FULLY-UPDATE for APASP instead of the DI update algorithm for APSP. Our change in the update sequence requires a completely new proof of the amortized bound which we present in Section 4. We consider this to be a contribution of independent interest: If we replace FULLY-UPDATE by the DI update algorithm in FULLY-DYNAMIC, we get a new fully dynamic APSP algorithm which

is as simple as DI, with a new analysis.

### 3 Algorithm FULLY-UPDATE

We first extend the notions of historical and locally historical paths [7] to tuples and triples.

**Definition 5** (HT, THT, LHT, and TLHT). *Let  $\tau$  be a tuple in the tuple system at time  $t$ . Let  $t' \leq t$  denote the most recent step at which a vertex on a path in  $\tau$  was updated. Then  $\tau$  is a historical tuple (HT) at time  $t$  if  $\tau$  was an ST-tuple at least once in the interval  $[t', t]$ ;  $\tau$  is a true HT (THT) at time  $t$  if it is not an ST in the current graph. A tuple  $\tau$  is a locally historical tuple (LHT) at time  $t$  if either it only contains a single vertex or every proper sub-path in it is an HT at time  $t$ ; a tuple  $\tau$  is a true LHT (TLHT) at time  $t$  if it is not an LST in the current graph.*

In FULLY-UPDATE,  $P^*(x, y)$  and  $P(x, y)$  will contain HTs (including all STs) and LHTs (including all LSTs), respectively, from  $x$  to  $y$ . Given an update to a vertex in the current graph, the algorithm FULLY-UPDATE (Algorithm 2 in the appendix) performs a call to a ‘cleanup’ routine, followed by a call to a ‘fixup’ routine. This is similar to the update procedure in the decremental and fully dynamic algorithms for unique paths in DI [7] and in the decremental algorithm in [19].

It was observed in [7] that the decremental algorithm they presented for the unique SP case is a correct algorithm when incremental updates are interleaved with decremental ones. However here, in contrast to DI, the decremental APASP algorithm in [19] (the NPR algorithm) needs to be refined before it becomes correct for a fully dynamic sequence, and this is due to the presence of multiple shortest paths. Consider, for instance, a THT  $\tau = (xa, by)$  with weight  $wt$  which is currently an LST. With the NPR algorithm [19], since  $\tau$  is a THT it will be present in  $P^*(x, y)$  (but not as an ST). Suppose a new set of paths represented by a new triple  $\tau' = ((xa, by), wt, count')$ , with the same weight  $wt$ , is added to the count of this tuple  $\tau$ . We cannot simply add  $count'$  to  $\tau$  in  $P^*(x, y)$  because its extensions were performed using the old count, and if  $\tau$  is restored as an ST in  $P^*(x, y)$ , these extensions will not have the correct count. (With unique SPs this situation can never occur.) Our solution here is to have  $\tau$  in  $P(x, y)$  with the larger correct count (thus including  $count'$ ), and to leave the corresponding  $\tau$  in  $P^*(x, y)$  with its original count. Should  $\tau$  later be restored as an ST then the difference in counts between  $\tau$  in  $P^*(x, y)$  and the corresponding  $\tau$  in  $P(x, y)$  will trigger left and right extensions of  $\tau$  with the correct count even though  $\tau$  is currently in  $P^*(x, y)$ .

There are additional subtleties. During FULLY-CLEANUP starting from the current updated vertex  $v$ , we may reach a triple  $\gamma = (\tau, wt, count)$  in  $P$  through say, a left extension, while the triple in  $P^*$  for  $\tau$  with weight  $wt$  is  $\gamma' = (\tau, wt, count')$ , with  $count' < count$  (the extreme case being that  $count' = 0$ , in which case there is no  $\gamma'$  in  $P^*$ ). This is an indication that the paths in  $\gamma - \gamma'$  were formed after  $\gamma'$  became a THT. We address this situation in FULLY-CLEANUP by noting that a THT contains obsolete information that can be completely removed without affecting the correctness of the algorithm. We apply the same approach to TLHTs, since these can also be removed without affecting correctness. Hence, when we encounter a THT or a TLHT during cleanup, we completely remove the triple from the tuple system.

The pseudocode for FULLY-CLEANUP and FULLY-FIXUP are given at the end of this paper (after the references) in Algorithm 3 and Algorithm 4, with the changes from the pseudocode for the decremental algorithm highlighted. Recall that FULLY-UPDATE (Algorithm 2) is simply an execution of FULLY-CLEANUP followed by FULLY-FIXUP. The correctness of this algorithm is argued by noting that our method ensures that when an ST in  $P^*$  is processed during FULLY-FIXUP without further extensions, it has the correct weight and count and all of its extensions have been performed with that count; every ST and LST is generated starting with singleton edges, min-weight tuples from the  $P$  sets, and correct STs from the  $P^*$  sets, hence the counts of the tuples identified as STs and LSTs are maintained correctly.

#### 3.1 Analysis and Properties of FULLY-UPDATE

The pseudocode for algorithm FULLY-UPDATE and its proof are given in Appendix A. In this section we establish some basic properties of algorithm FULLY-UPDATE based on the high-level description we have

given above.

Let  $C$  be the maximum number of tuples in the tuple system containing a path through a given vertex at any time, and let  $D$  be the maximum number of tuples that can be in the tuple system at any time. For the complexity analysis, we observe that any triple examined during the cleanup phase contains at least one path going through the updated node  $v$ . Thus the number of triples examined during the cleanup phase cannot exceed  $C$  in the worst case.

The amortized time bound for FULLY-FIXUP is obtained similar to NPR [19]. This analysis observes that many existing tuples may simply have their count increased, but any such tuple contains a path through the updated vertex  $v$ , and hence their number is bounded by  $O(C)$ . Thus, during a call to FULLY-FIXUP, the number of existing tuples examined is  $O(n^2 + C)$ , and any other tuple examined is newly created during the FULLY-FIXUP step or moved from  $P$  to  $P^*$ . Further, FULLY-FIXUP does not remove any tuple. Hence, the total number of tuples accessed in all FULLY-FIXUP steps across a sequence of  $r$  updates is bounded by  $O(r \cdot (n^2 + C))$  for the existing tuples examined, plus the total number of tuples  $T$  added to the tuple system by FULLY-FIXUP. This number  $T$  is no more than the number of tuples removed by all FULLY-CLEANUP steps,  $O(r \cdot C)$ , plus  $D$ , the number of tuples that can remain in the tuple system at the end of the update sequence. Factoring in the  $O(\log n)$  cost for accessing each tuple in the data structures, we obtain the following lemma (proof in Appendix B).

**Lemma 6.** *Consider a sequence of  $r$  calls to FULLY-UPDATE on a graph with  $n$  vertices. Let  $C$  be the maximum number of tuples in the tuple system that can contain a path through a given vertex, and let  $D$  be the maximum number of tuples that can be in the tuple system at any time. Then FULLY-UPDATE executes the  $r$  updates in  $O((r \cdot (n^2 + C) + D) \cdot \log n)$  time.*

In Section 4, we will use the algorithm FULLY-UPDATE within algorithm FULLY-DYNAMIC, that performs a special sequence of ‘dummy’ updates, to obtain a fully dynamic APASP algorithm with an  $O(\nu^{*2} \log^3 n)$  amortized cost per update; we obtain this amortized bound by establishing suitable upper bounds on the parameters  $C$  and  $D$  in Lemma 6 when algorithm FULLY-DYNAMIC is used. We conclude this section with some lemmas that will be used in the analysis of the amortized time bound of Algorithm 1. The proof of the following lemma is fairly straightforward, but since it refers to the pseudocode in Appendix A, it is deferred to Appendix B.

**Lemma 7.** *At each step  $t$ , the tuple system for Algorithm FULLY-UPDATE maintains a subset of HTs and LHTs that includes all STs and LSTs for step  $t$ , and further, for every LHT triple  $((x, y), wt, count)$  in step  $t$ , there are HTs  $(a, y)$  and  $(x, b)$  with weights  $wt - w(x, a)$  and  $wt - w(b, y)$  respectively, in that step.*

**Lemma 8.** *If FULLY-UPDATE is called on vertex  $v$  at step  $t$ , then at the end of step  $t$ :*

1. *There is no THT in the tuple system that contains a path through  $v$ .*
2. *For any TLHT  $(x, y)$  in the tuple system that contains a path through  $v$ , the vertex  $v$  must be one of the endpoints  $x, y$ .*

*Proof.* Part 1 of the lemma follows from Definition 5. For part 2, by Lemma 7, any LHT in the tuple system is formed by combining two HTs that are in the tuple system. Now consider the TLHT  $(x, y)$  that contains  $v$ . Since by definition of TLHT,  $(x, y)$  is not an LST in the current graph, assume w.l.o.g. that  $(x, b)$  is the subtuple that is not an ST when an end edge is deleted from  $(x, y)$ . But this is not possible since any tuple HT  $(x, b)$  that contains  $v$  must be an ST. The lemma follows.  $\square$

**Lemma 9.** *Let  $G$  be a graph after a sequence of calls to FULLY-UPDATE, and suppose every HT in the tuple system is an ST in one of  $z$  different graphs  $H_1, \dots, H_z$ , and every LHT is formed from these HTs. If  $n$  and  $m$  bound the number of vertices and edges, respectively, in any of these graphs, and if  $\nu^*$  bounds the maximum number of edges that lie on shortest paths through any given vertex in any of these graphs, then:*

1. The number of LHTs in  $G$ 's tuple system is at most  $O(z \cdot m \cdot \nu^*)$ .
2. The number of LHTs that contain the newly updated vertex  $v$  in  $G$  is  $O(z \cdot \nu^{*2})$ .
3. Let  $u$  be any vertex in  $G$ , and suppose that the HTs that contain  $u$  lie in  $z' \leq z$  of the graphs  $H_1, \dots, H_z$ . Then, the number of LHTs that contain the vertex  $u$  is  $O((z + z'^2) \cdot \nu^{*2})$ .

*Proof.* For part 1, we bound the number of LHTs  $(xa, by)$  (across all weights) that can exist in  $G$ . The edge  $(x, a)$  can be chosen in  $m$  ways, and once we fix  $(x, a)$ , the  $r$ -tuple  $(a, by)$  must be an ST in one of the  $H_j$ . Since  $(b, y)$  must lie on a shortest path through  $a$  in the graph  $H_i$  that contains the  $r$ -tuple  $(a, by)$  of that weight, the number of different choices for  $(b, y)$  that will then uniquely determine the tuple  $(xa, by)$ , together with its weight, is  $z \cdot \nu^*$ . Hence the number of LHTs in  $G$ 's tuple system is  $O(z \cdot m \cdot \nu^*)$ .

For part 2, the number of LHTs that contain  $v$  as an internal vertex is simply the number of LSTs in the current graph by Lemma 8 (part 2), and using Lemma 4, this is  $O(\nu^{*2})$ . We now bound the number of LHTs  $(va, by)$ . There are  $n - 1$  choices for the edge  $(v, a)$  and  $z \cdot \nu^*$  choices for the  $r$ -tuple  $(a, by)$ , hence the total number of such tuples is  $O(z \cdot n \cdot \nu^*)$ . The same bound holds for LHTs of the form  $(xa, bv)$ . Since  $\nu^* = \Omega(n)$ , the result in part 2 follows.

For part 3, we observe that each LHT that contains  $u$  as an internal vertex must be composed of two HTs, each of which is an ST in one of the  $z'$  graphs that contain  $v$ . Thus, there are  $O(z'^2 \cdot \nu^{*2})$  such tuples. For an LHT, say  $\tau = (ua, by)$ , that contains  $u$  as an end vertex, the analysis remains the same as above in part 2: there are  $n - 1$  choices for the edge  $(u, a)$  and  $z \cdot \nu^*$  choices for the  $r$ -tuple  $(a, by)$ , hence the total number of such tuples is  $O(z \cdot n \cdot \nu^*)$ . This gives the desired result.  $\square$

## 4 The Overall Algorithm FULLY-DYNAMIC

Algorithm FULLY-UPDATE in Section 3 is a correct fully dynamic algorithm for APASP, but it is not a very efficient algorithm, since  $C$  and  $D$  in Lemma 6 could be very large. We now present our overall fully dynamic algorithm for APASP. As in [7, 27] we build up the tuple system for the initial  $n$ -node graph  $G = (V, E)$  with  $n$  inserts starting with the empty graph (and hence  $n$  incremental updates), and we then perform the first  $n$  updates in the given update sequence  $\Sigma$ . After these  $2n$  updates, we reset all data structures and start afresh.

Consider a graph  $G = (V, E)$  with weight function  $\mathbf{w}$  in which an incremental or decremental update is applied to a vertex  $u$ . Let  $\mathbf{w}'$  be the weight function after the update, hence the only changes to the edge weights occur on edges incident to  $u$ . Algorithm 1 gives the overall fully dynamic algorithm for the  $t$ -th update to a vertex  $v$  with the new weight function  $\mathbf{w}'$ . This algorithm applies FULLY-UPDATE to vertex  $v$  with the new weight function, thus it will be correct if we executed only the first step, but not necessarily efficient. To obtain an efficient algorithm we execute 'dummy updates' on a sequence  $\mathcal{N}$  of the most recently updated vertices as specified in Steps 2-5. The length of this sequence of vertices is determined by the position  $k$  of the least significant bit set to 1 in the bit representation  $B = b_{r-1} \dots b_0$  of  $t$ . We denote  $k$  by  $set-bit(t)$ .

---

### Algorithm 1 FULLY-DYNAMIC( $G, v, \mathbf{w}', t$ )

---

- 1: FULLY-UPDATE( $v, \mathbf{w}'$ )
  - 2:  $k \leftarrow set-bit(t)$  (i.e., if the bit representation of  $t$  is  $b_{r-1} \dots b_0$ , then  $b_k$  is the least significant bit with value 1)
  - 3:  $\mathcal{N} \leftarrow$  set of vertices updated at steps  $t - 1, \dots, t - (2^k - 1)$
  - 4: **for** each  $u \in \mathcal{N}$  in decreasing order of update time **do**
  - 5:     FULLY-UPDATE( $u, \mathbf{w}'$ )     (dummy updates)
- 

**Properties of  $\mathcal{N}$ .** Consider the current update step  $t$ , with its bit representation  $B = b_{r-1} \dots b_0$  and with  $set-bit(t) = k$ . We say that index  $i$  is a *time-stamp* for  $t$  if  $b_i = 1$  (so  $r - 1 \geq i \geq k$ ), and for each such



time-stamp  $i$ , we let  $time_t(i)$  be the earlier update step  $t'$  whose bit representation has zeros in positions  $b_i - 1, \dots, 0$ , and which matches  $B$  in positions  $b_{r-1} \dots b_i$ . In other words,  $time_t(i) = t'$ , where  $t'$  has bit representation  $b_{r-1} \dots b_{i+1} b_i 0 \dots 0$ . We do not define  $time_t(i)$  if  $b_i = 0$ . We define  $Prior-times(t)$  be the set of  $time_t(i)$ , where  $i$  is a time-stamp for  $t$ . Note that  $|Prior-times(t)| \leq r = O(\log n)$ , since  $r \leq \log(2n)$ .

Let  $G_t$  be the graph after the  $t$ -th update is applied,  $t \geq 1$ , with the initial graph being  $G_0$ . Thus, the input graph to Algorithm 1 is  $G_{t-1}$ , and the updated graph is  $G_t$ .

**Lemma 10.** *For every vertex  $v$  in  $G_t$ , the step  $t_v$  of the most recent update to  $v$  is in  $Prior-times(t)$ .*

*Proof.* Let the bit representation of  $t$  be  $B = b_{r-1} \dots b_0$ , let  $set-bit(t) = i$ , and let  $j_1 > j_2 > \dots > j_s = i$  be the time-stamps for  $t$ . Let  $t_u = time_t(b_{j_u})$ , thus the bit representation of  $t_u$  is the same as  $B$ , with all bits in positions less than  $j_u$  set to zero; let  $t_0 = 1$ . Then, we observe that during the execution of Algorithm 1 for the  $t_u$ -th update, the vertex for update  $t_u$  will be updated in Step 1, and the vertices updated in steps  $t_{u-1} + 1, \dots, t_u - 1$  will be updated in Steps 4-5 of Algorithm 1. Hence all vertices updated in  $[t_{u-1} + 1, t_u]$  are more recently updated in step  $t_u$ . Thus the most recent update step for every vertex in  $G_t$  is one of the  $O(\log n)$  steps in  $Prior-times(t)$ .  $\square$

## 5 Analysis of Algorithm FULLY-DYNAMIC

The analysis in this section incorporates many elements from Thorup's algorithm [27]. However, these are present only in the analysis, and the only component of that rather complicated algorithm that we use is the form of the dummy update sequence in Step 3 of Algorithm 1. Our Algorithm 1 is about as simple as the DI algorithm when specialized to unique shortest paths since in that case it suffices to use the update algorithm in DI instead of the more elaborate FULLY-UPDATE we use here.

### 5.1 The Prior Deletion Graph (PDG)

Let  $t' < t$  be two update steps, and let  $W$  be the set of vertices that are updated in the interval of steps  $[t' + 1, t]$ . We define the *prior deletion graph (PDG)*  $\Gamma_{t',t}$  as the induced subgraph of  $G_{t'}$  on the vertex set  $V(G_{t'}) - W$ . If  $t$  is the current update step, then we simply use  $\Gamma_{t'}$  instead of  $\Gamma_{t',t}$ .

We say that a path  $p$  is *present in both  $G_{t'}$  and  $G_t$*  if no call to FULLY-UPDATE is made on any vertex in  $p$  during the update steps in the interval  $[t' + 1, t]$ .

**Lemma 11.** *Let tuple  $\tau$  represent a collection of paths in  $G_{t'}$ . Then,*

1. *If  $\tau$  is an ST in  $G_{t'}$  then  $\tau$  continues to be an ST in every PDG  $\Gamma_{t',t}$  with  $t \geq t'$  in which  $\tau$  is present.*
2. *For any  $\hat{t} \geq t'$ , if  $\tau$  is an ST in  $G_{\hat{t}}$  then  $\tau$  is an ST in every PDG  $\Gamma_{t',t''}$ ,  $t'' \geq \hat{t}$ , in which  $\tau$  is present.*

*Proof.* As observed in [19] (and in [7] for unique shortest paths), an ST in a graph remains an ST after a weight increase on any edge that is not on it. This establishes the first part. For the second part, we observe that  $\Gamma_{t',\hat{t}}$  can be viewed as being obtained from  $G_{\hat{t}}$  by deleting the vertices updated in  $[t' + 1, \hat{t}]$ . Since  $\tau$  is an ST in  $G_{\hat{t}}$ , it continues to be an ST in the graph  $\Gamma_{t',\hat{t}}$ , which can be obtained from  $G_{\hat{t}}$  through a sequence of decremental updates that do not change the weight of any edge on  $\tau$ . Finally since  $\tau$  is an ST in  $\Gamma_{t',\hat{t}}$ , it must be an ST in any  $\Gamma_{t',t''}$  in which it appears, for  $t'' > \hat{t}$ .  $\square$

**PDGs for Update  $t$ :** We will associate with the current update step  $t$ , the set of PDGs  $\Gamma_{t'}$ , for  $t' \in Prior-times(t)$ . These PDGs are similar to the *level graphs* maintained in Thorup's algorithm [27], but we choose to give them a different name since we use them here only to analyze the performance of our algorithm.

**Lemma 12.** *Consider a sequence of fully dynamic updates performed using Algorithm 1. Let the current update step be  $t$  and consider the set of graphs  $\Gamma_{t'}$ , for  $t' \in Prior-times(t)$ . Then, each HT in the tuple system for  $G_t$  is an ST in at least one of the  $\Gamma_{t'} \in Prior-times(t)$ . Further  $z = O(\log n)$  in Lemma 9 for  $G_t$ .*

*Proof.* Consider an HT  $\tau = (xa, by)$  in  $G_t$ . Let the most recently updated vertex in  $\tau$  be  $v$ , and let its update step be  $t_v \leq t$ . By definition of HT,  $\tau$  is an ST in some  $t'$  in  $[t_v, t]$ , and hence by part 2 of Lemma 11, using  $\hat{t} = t' = t_v$  and  $t'' = t$ ,  $\tau$  is an ST in  $\Gamma_{t_v}$ . By Lemma 10,  $t_v \in \text{Prior-times}(t)$ . Finally, since  $|\text{Prior-times}(t)| \leq \log n$  for any  $t$ ,  $z = O(\log n)$  in Lemma 9.  $\square$

We will use the above lemma to obtain our amortized time bound in Lemma 13 in the next section. It is not clear that a similar result can be obtained with the DI dummy update sequence.

## 5.2 Amortized Cost of Algorithm FULLY-DYNAMIC

We will now bound the amortized cost of an update in a sequence  $\Sigma$  of  $n$  real updates on an initial  $n$ -node graph  $G = (V, E)$ . As mentioned earlier, in our method this will translate to a sequence  $\Sigma'$  of  $2n$  real (as opposed to dummy) updates: there is an initial sequence of  $n$  updates, starting with the empty graph, which inserts each of the  $n$  vertices in  $G$  along with incident edges that have not yet been inserted. Following this is the sequence of  $n$  real updates in  $\Sigma$ . Each of these  $2n$  updates in  $\Sigma'$  will make a call to Algorithm 1. As before, let  $\nu^*$  be a bound on the number of edges that lie on shortest paths through any given vertex in any of  $G_t$ . The following lemma establishes our main Theorem 1.

**Lemma 13.** *Algorithm 1 executes a sequence  $\Sigma$  of  $n$  real updates on an  $n$ -node graph in  $O(\nu^{*2} \cdot \log^3 n)$  amortized time per update.*

*Proof.* Let  $n' = 2n$ , and as described above, let  $\Sigma'$  be the sequence of  $n'$  calls to Algorithm 1 used to execute the  $n$  updates in  $\Sigma$ . We first observe that Algorithm 1 performs  $O(n' \log n')$  dummy updates when executing these  $n'$  calls. This is because there are  $n'/2^k$  real updates for update steps  $t$  with  $\text{set-bit}(t) = k$ , and each such update is accompanied by  $2^k - 1$  dummy updates. So, across all real updates there are  $O(n')$  dummy updates for each position of  $\text{set-bit}$ , adding up to  $O(n' \log n')$  in total, across all  $\text{set-bit}$  positions.

We now use Lemma 6 to bound the time needed to execute the  $d = n' \log n'$  dummy updates and  $n'$  real updates. We need to bound the parameters  $C$  and  $D$  in Lemma 6. We first consider  $D$ . For this we will use part 1 of Lemma 9. By Lemma 12,  $z = O(\log n)$  for any  $t$ . Hence by Lemma 9 the maximum number of tuples that can remain at the end of the update sequence is  $D = O(m \cdot \nu^* \cdot \log n)$ .

Now we bound the parameter  $C$ , for which we will obtain separate bounds,  $C_1$  for the real updates, and  $C_2$  for the dummy updates. For each real update we have  $z = z' = O(\log n)$  in part 3 of Lemma 9, hence the number of tuples that contain a path through the updated vertex is  $O(\nu^{*2} \cdot \log^2 n)$ , thus  $C_1 = O(\nu^{*2} \cdot \log^2 n)$ .

Now consider a dummy update on a vertex  $u$  in Step 5 of Algorithm 1, and let  $u$  be in some level  $i$  (note the  $i$  must be less than the current level  $k$ ). At the time this call is made, all vertices that were updated after  $u$  was last updated in the graph (i.e., after  $\text{time}(i)$ ) are now in the newest level  $k$ . Thus, any LHT that contains  $u$  lies in either  $\Gamma_{\text{time}(i)}$  or in  $G_t = \Gamma_t$ . Hence  $z' = 2$  in part 3 of Lemma 9 for any vertex  $u$  undergoing a dummy update. Thus  $C_2 = O(\nu^{*2} \cdot \log n)$ .

Hence the total time taken by Algorithm 1 for its  $d = n' \log n'$  dummy updates and the  $n'$  real updates is, by Lemma 6,  $O((n' \cdot (n^2 + C_1) + (n' \log n') \cdot (n^2 + C_2) + D) \cdot \log n) = O(n' \cdot \nu^{*2} \cdot \log^2 n + (n' \cdot \log n') \cdot \nu^{*2} \cdot \log n + D \log n) = O(n \cdot \nu^{*2} \cdot \log^3 n)$  (the  $n^2$  and  $D$  terms are dropped since  $\nu^* = \Omega(n)$ , hence  $m = O(n \cdot \nu^*)$ ).

It follows that the amortized cost of each of the  $n$  updates in  $\Sigma$  is  $O(\frac{1}{n} \cdot n' \cdot \nu^{*2} \cdot \log^3 n) = O(\nu^{*2} \cdot \log^3 n)$ .  $\square$

## 6 Conclusion

We have presented an efficient fully dynamic algorithm for APASP. Our algorithm stores a superset of the STs and LSTs in the current graph in two priority queues  $P^*(x, y)$  and  $P(x, y)$  for each vertex pair  $x, y$ . To generate all shortest paths from all other vertices to  $x$ , we construct the shortest path in-dag rooted at  $x$  in  $O(\nu^*)$  time. Then, the shortest paths ending in  $x$  can be enumerated in a traversal of this dag, starting from  $x$ . This query time is output-optimal, and takes time proportional to the number of edges on these paths.

Our algorithm, when specialized to fully dynamic APSP (i.e., for unique SPs), is a variant of the DI [7] method, and it uses a different ‘dummy update sequence’ from the one in [7], with different properties. Our

dummy update sequence is inspired by the updates performed on ‘level graphs’ in [27], though our algorithm is considerably simpler (but is also slower by a logarithmic factor). As noted in Section 5.1, our analysis is tailored to the dummy update sequence we use, and a different analysis would be needed if the DI update sequence is to be used.

Recently we have developed a fully dynamic algorithm for APASP [21] that runs in amortized  $O(\nu^{*2} \cdot \log^2 n)$  time, which is a log factor faster than the result presented here. This algorithm is considerably more involved and adapts the method in Thorup [27] to the APASP problem by maintaining the PDGs explicitly. An additional complexity in this fully dynamic APASP algorithm (beyond that present in [27]) is the need to maintain several different time-stamps for each tuple in the data structures, since the component paths in a tuple may have been updated at different time steps.

It would be interesting to investigate if one could stay with our method here of only performing dummy updates and not maintaining the PDGs explicitly, but still obtain the improved bound in our companion paper [21] (and in Thorup [27] for unique shortest paths). This would give a reasonably simple fully dynamic APASP algorithm that is a logarithmic factor faster than the algorithm in this paper, and it would lead to a simpler fully dynamic APSP algorithm with  $O(n^2 \cdot \log^2 n)$  amortized time than Thorup’s algorithm [27]. One appealing approach is to only form LSTs in the current graph during FULLY-FIXUP (instead of forming all LHTs in the for loop starting on line 24 of FULLY-FIXUP). It is not difficult to see that this would reduce the cost of FULLY-CLEANUP by a logarithmic factor. However, if an HT  $\tau$  becomes an ST at a later step, we would then have no guarantee that all of its extensions have been generated. Hence, if this approach is to succeed, a modified algorithm is needed.

**Acknowledgement:** The authors would like to thank the reviewers for their comments, which helped to improve the presentation of the write-up.

## References

- [1] Hans-Jrgen Bandelt and Henry Martyn Mulder. Interval-regular graphs of diameter two. *Discrete Mathematics*, 50(0):117 – 134, 1984.
- [2] Elisabetta Bergamini and Henning Meyerhenke. Fully-dynamic approximation of betweenness centrality. arXiv:1504.07091 [cs.DS], 2015.
- [3] Elisabetta Bergamini, Henning Meyerhenke, and Christian L. Staudt. Approximating betweenness centrality in large evolving networks. In *Proc. of ALENEX 2015*, chapter 11, pages 133–146. SIAM, 2015.
- [4] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
- [5] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, 2004.
- [6] Anne Condon and Richard M. Karp. Algorithms for graph partitioning on the planted partition model. *Random Struct. Algorithms*, 18(2):116–140, March 2001.
- [7] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
- [8] Camil Demetrescu and Giuseppe F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Trans. Algorithms (TALG)*, 2(64):578–601, 2006.
- [9] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *Proc. of ALENEX 2008*, chapter 8, pages 90–100. SIAM, 2008.

- [10] Oded Green, Robert McColl, and David A. Bader. A fast algorithm for streaming betweenness centrality. In *Proc. of 4th PASSAT*, pages 11–20, 2012.
- [11] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993.
- [12] Miray Kas, Matthew Wachs, Kathleen M. Carley, and L. Richard Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *Proc. of ASONAM*, 2013.
- [13] Nicolas Kourtellis, Tharaka Alahakoon, Ramanuja Simha, Adriana Iamnitchi, and Rahul Tripathi. Identifying high betweenness centrality nodes in large social networks. *SNAM*, pages 1–16, 2012.
- [14] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. Scalable online betweenness centrality in evolving graphs. *IEEE Trans. Knowl. Data Eng.*, 27(9):2494–2506, 2015.
- [15] Valdis Krebs. Mapping networks of terrorist cells. *CONNECTIONS*, 24(3):43–52, 2002.
- [16] Min-Joong Lee, Jungmin Lee, Jaimie Yejean Park, Ryan Hyun Choi, and Chin-Wan Chung. Qube: a quick algorithm for updating betweenness centrality. In *Proc. 21st WWW Conference*, pages 351–360, 2012.
- [17] Henry Martyn Mulder. Interval-regular graphs. *Discrete Mathematics*, 41(3):253 – 269, 1982.
- [18] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Betweenness centrality incremental and faster. In *MFCS 2014*, volume 8635 of *LNCS*, pages 577–588. Springer, 2014.
- [19] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Decremental all-pairs all shortest paths and betweenness centrality. In *ISAAC 2014*, volume 8889 of *LNCS*, pages 766–778. Springer, 2014.
- [20] John W. Pinney, Gleen A. McConkey, and David R. Westhead. Decomposition of biological networks using betweenness centrality. In *Proc. of 9th RECOMB*, 2005.
- [21] Matteo Pontecorvi and Vijaya Ramachandran. A faster algorithm for fully dynamic betweenness centrality. <http://arxiv.org/abs/1506.05783>, 2015.
- [22] J. Sicilia R. M. Ramos and M. T. Ramos. A generalization of geodetic graphs: K-geodetic graphs. *Inverstigacin Operativa*, 1:85–101, 1998.
- [23] Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proc. of the 7th ACM WSDM*, pages 413–422. ACM, 2014.
- [24] Satu Elisa Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, August 2007.
- [25] Rishi Ranjan Singh, Keshav Goel, Sudarshan Iyengar, and Sukrit. A faster algorithm to update betweenness centrality after node alteration. In *Proc. of 10th WAW*, 2013.
- [26] N. Srinivasan, J. Opatrny, and V.S. Alagar. Bigeodetic graphs. *Graphs and Combinatorics*, 4(1):379–392, 1988.
- [27] Mikkel Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT*, pages 384–396, 2004.

## A Pseudocode for Algorithm FULLY-UPDATE

FULLY-UPDATE is given in the two-line Algorithm 2. It calls Algorithm 3 (FULLY-CLEANUP) and Algorithm 4 (FULLY-FIXUP), in sequence, on the updated vertex  $v$ . We present pseudocode for these two procedures here. They are similar to the corresponding pseudocode cleanup and fixup in [19], and we have marked the steps changed from [19] with a  $\bullet$  at the end of the line. Also, a triple  $\gamma$  is now inserted in  $P$  with a key  $[wt, \beta]$ , instead of just  $wt$ . Here  $\beta$  is a control bit that is set during the fixup phase and indicates if  $\gamma$  is present in  $P^*$  with the correct count ( $\beta = 1$ ), or if  $\gamma$  has the correct count only in  $P$  ( $\beta = 0$ ). In the latter case  $\gamma$  could be also present in  $P^*$  but with a wrong count.

The only changes from [19] in Algorithm 3 are in steps 11, 18, 23 and 26 where we remove any THT or TLHT we encounter during cleanup while extending from the updated node  $v$ . More specifically, a triple  $(x'x, by)$  with weight  $wt'$  is an LST iff  $wt' - \mathbf{w}(b, y) = d(x', b)$  (as checked in step 11, Algorithm 3), and in this case we proceed as in the decremental-only algorithm; otherwise the triple is a TLHT and we remove it completely from the tuple system (Steps 18–20, Algorithm 3). The same method applies to HTs (steps 23, 26, Algorithm 3). Steps 11 and 23 are performed using the shortest distances between any two nodes before the cleanup phase starts. The other steps in Algorithm 3 are described in [19].

Algorithm 4 introduces new features to achieve correctness and efficiency. We observe that we may revert an HT from, say,  $x$  to  $y$ , back to an ST during an update, and this happens only if the shortest path distance from  $x$  to  $y$  increases. This condition translates into the new check in Step 9 of Algorithm 4. Here we proceed as in [19] keeping in mind that an LHT extracted from  $P$  as an ST (Step 10, Algorithm 4) may or may not be in  $P^*$ . If the LHT is not in  $P^*$  (Step 13, Algorithm 4) we add the triple to the tuple system as in [19]. If the LHT is already in  $P^*$  with a different count (Step 16, Algorithm 4), we replace the count of the triple in  $P^*$  with  $count'$  from the triple in  $P$  and we add the triple to  $S$  (Step 17, Algorithm 4). Note that the extension sets  $L^*$  and  $R^*$  do not need to be updated in this case. In step 10, Algorithm 4 we check the bit  $\beta$  associated with the triple  $\gamma$ . Since  $P$  is a priority queue, we will process only the triples in  $P$  with min-key  $[wt, 0]$ , so we avoid examining the triples that are already in  $P^*$  with a correct count. We set  $\beta$  to 1 for any triple added to or updated in  $P^*$  with the correct count (Steps 18 and 23, Algorithm 4). Also, for an LHT updated in  $P$  and not  $P^*$ , we set  $\beta = 0$  (Step 34, Algorithm 4). The other steps in Algorithm 4 are described in [19], as are the two parameters  $paths(\gamma, v)$ , which gives the number of paths containing the node  $v$  that are represented by the triple  $\gamma$ , and  $update\_num(\gamma)$ , which is a timestamp that indicates the last update in which the triple  $\gamma$  is involved.

### A.1 Correctness

In this section, we establish the correctness of Algorithm 3 and 4. We assume that all the local structures are correct before the update, and we will show the correctness of them after the update. FULLY-CLEANUP works with a heap  $H_c$  of triples. The algorithm maintains the loop invariant that any triple inserted into  $H_c$  has already been deleted from the tuple system: only its extensions remain to be processed and deleted. We prove the following lemma:

**Lemma 14.** *After Algorithm 3 (FULLY-CLEANUP) is executed, for any  $(x, y) \in V$ , the STs in  $P^*(x, y)$  (LSTs in  $P(x, y)$ ) represent all the SPs (LSPs) from  $x$  to  $y$  in  $G$  that do not pass through  $v$ . Moreover, every THT (TLHT) present in the tuple system represents a collection of HPs (LHPs) in  $G$  that contains only paths that do not pass through  $v$ . Finally, the sets  $L, L^*, R, R^*$  are correctly maintained.*

*Proof.* To prove the lemma statement, we use a loop invariant on the while loop in Step 3 of Algorithm 3. We show that the while loop maintains the following invariants.

**Loop Invariant:** At the start of each iteration of the while loop in Step 3 of Algorithm 3, assume that the first triple to be extracted from  $H_c$  and processed has min-key =  $[wt, x, y]$ . Then the following properties hold about the tuple system and  $H_c$ .

1. For any  $a, b \in V$ , if  $G$  contains  $c_{ab}$  LHPs of weight  $wt$  of the form  $(xa, by)$  passing through  $v$ , then  $H_c$  contains a triple  $\gamma = ((xa, by), wt, c_{ab})$  with key  $[wt, x, y]$  already processed: the  $c_{ab}$  LHPs through  $v$  are cleaned from the system.

---

**Algorithm 2** FULLY-UPDATE( $v, \mathbf{w}'$ )

---

2. 1: FULLY-CLEANUP( $v$ )  
2: FULLY-FIXUP( $v, \mathbf{w}'$ )
- 

---

**Algorithm 3** FULLY-CLEANUP( $v$ )

---

```
1:  $H_c \leftarrow \emptyset$ ; Marked-Tuples  $\leftarrow \emptyset$ 
2:  $\gamma \leftarrow ((vv, vv), 0, 1)$ ; add  $\gamma$  to  $H_c$ 
3: while  $H_c \neq \emptyset$  do
4:   extract in  $S$  all the triples with min-key  $[wt, x, y]$  from  $H_c$ 
5:   for every  $b$  such that  $(x \times, by) \in S$  do
6:     let  $fcount' = \sum_i ct_i$  such that  $((xa_i, by), wt, ct_i) \in S$ 
7:     for every  $x' \in L(x, by)$  such that  $(x'x, by) \notin$  Marked-Tuples do
8:        $wt' \leftarrow wt + \mathbf{w}(x', x)$ ;  $\gamma' \leftarrow ((x'x, by), wt', fcount')$ ;
9:       add  $\gamma'$  to  $H_c$ 
10:      {The next step will check if  $\gamma'$  is an LST or a TLHT}
11:      if  $wt' - \mathbf{w}(b, y) = d(x', b)$  then
12:        remove  $\gamma'$  in  $P(x', y)$  // decrements its count in  $P$  by  $fcount'$  {we decrement an LST}
13:        if a triple for  $(x'x, by)$  exists in  $P(x', y)$  then
14:          insert  $(x'x, by)$  in Marked-Tuples
15:        else
16:          delete  $x'$  from  $L(x, by)$  and delete  $y$  from  $R(x'x, b)$ 
17:        else
18:          remove the tuple  $(x'x, by)$  with weight  $wt'$  from  $P(x', y)$  • {we completely remove a TLHT}
19:          if a triple for  $(x'x, by)$  does not exist in  $P(x', y)$  then
20:            delete  $x'$  from  $L(x, by)$  and delete  $y$  from  $R(x'x, b)$ 
21:          if a triple  $\gamma''$  for  $(x'x, by)$  exists in  $P^*(x', y)$  with weight  $wt'$  and  $count$  paths then
22:            {The next step will check if  $\gamma'$  is an ST or a THT}
23:            if  $wt' = d(x', y)$  then
24:              remove  $\gamma'$  in  $P^*(x', y)$  // decrements  $count$  by  $fcount'$  {we decrement an ST}
25:            else
26:              remove  $\gamma''$  from  $P^*(x', y)$  • {we completely remove a THT}
27:            if  $P^*(x, y) = \emptyset$  then delete  $x'$  from  $L^*(x, y)$ 
28:            if  $P^*(x', b) = \emptyset$  then delete  $y$  from  $R^*(x', b)$ 
29:      perform symmetric steps 5 – 28 for right extensions
```

---

---

**Algorithm 4** FULLY-FIXUP( $v, \mathbf{w}'$ )

---

```
1:  $H_f \leftarrow \emptyset$ ; Marked-Tuples  $\leftarrow \emptyset$ 
2: for each edge incident on  $v$  do
3:   create a triple  $\gamma$ ; set  $paths(\gamma, v) = 1$ ; set  $update\_num(\gamma)$ ; add  $\gamma$  to  $H_f$  and to  $P()$ 
4: for each  $x, y \in V$  do
5:   add a min-key triple from  $P(x, y)$  to  $H_f$ 
6: while  $H_f \neq \emptyset$  do
7:   extract in  $S'$  all triples with min-key  $[wt, x, y]$  from  $H_f$ ;  $S \leftarrow \emptyset$ 
8:   if  $S'$  is the first extracted set from  $H_f$  for  $x, y$  then
9:     if  $P^*(x, y)$  increased min-weight after cleanup then
10:      for each  $\gamma' \in P(x, y)$  with min-key  $[wt, 0]$  do
11:        let  $\gamma' = ((xa', by), wt, count')$ 
12:        {Next step check if  $\gamma'$  is completely missing from  $P^*$ }
13:        if  $\gamma'$  is not in  $P^*(x, y)$  then
14:          add  $\gamma'$  in  $P^*(x, y)$  and  $S$ ; add  $x$  to  $L^*(a', y)$  and  $y$  to  $R^*(x, b')$ 
15:          {Next step check if  $\gamma'$  is in  $P^*$  with a different count}
16:        else if  $\gamma'$  is in  $P(x, y)$  and  $P^*(x, y)$  with different counts then
17:          replace the count of  $\gamma'$  in  $P^*(x, y)$  with  $count'$  and add  $\gamma'$  to  $S$  •
18:        set  $\beta$  for  $\gamma' \in P(x', y)$  to 1 •
19:      else
20:        for each  $\gamma' \in S'$  containing a path through  $v$  do
21:          let  $\gamma' = ((xa', b'y), wt, count')$ 
22:          add  $\gamma'$  with  $paths(\gamma', v)$  in  $P^*(x, y)$  and  $S$ ; add  $x$  to  $L^*(a', y)$  and  $y$  to  $R^*(x, b')$ 
23:          set  $\beta$  for  $\gamma' \in P(x', y)$  to 1 •
24:      for every  $b$  such that  $(x \times, by) \in S$  do
25:        let  $fcount' = \sum_i ct_i$  such that  $((xa_i, by), wt, ct_i) \in S$ 
26:        for every  $x'$  in  $L^*(x, b)$  do
27:          if  $(x'x, by) \notin$  Marked-Tuples then
28:             $wt' \leftarrow wt + \mathbf{w}(x', x)$ ;  $\gamma' \leftarrow ((x'x, by), wt', fcount')$ 
29:            set  $update\_num(\gamma')$ ;  $paths(\gamma', v) \leftarrow \sum_{\gamma=(x \times, by)} paths(\gamma, v)$ ; add  $\gamma'$  to  $H_f$ 
30:            if a triple for  $((x'x, by), wt')$  exists in  $P(x', y)$  then
31:              add  $\gamma'$  with  $paths(\gamma', v)$  in  $P(x', y)$ ; add  $(x'x, by)$  to Marked-Tuples
32:            else
33:              add  $\gamma'$  to  $P(x', y)$ ; add  $x'$  to  $L(x, by)$  and  $y$  to  $R(x'x, b)$ 
34:            set  $\beta$  for  $\gamma' \in P(x', y)$  to 0 •
35:      perform symmetric steps 24 – 34 for right extensions
```

---

Let  $[\hat{wt}, \hat{x}, \hat{y}]$  be the last key extracted from  $H_c$  and processed before  $[wt, x, y]$ . For any key  $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$ , let  $G$  contain  $c > 0$  number of LHPs of weight  $wt_1$  of the form  $(x_1 \times, b_1 y_1)$ . Further, let  $c_v$  (resp.  $c_{\bar{v}}$ ) denote the number of such LHPs that pass through  $v$  (resp. do not pass through  $v$ ). Here  $c_v + c_{\bar{v}} = c$ . For every extension  $x' \in L(x_1, b_1 y_1)$ , let  $wt' = wt_1 + \mathbf{w}(x', x_1)$  be the weight of the extended triple  $(x' x_1, b_1 y_1)$ . Then, (the following assertions are similar for  $y' \in R(x_1 a_1, y_1)$ )

- (a) if  $c > c_v$  there is a triple in  $P(x', y_1)$  of the form  $(x' x_1, b_1 y_1)$  and weight  $wt'$  representing  $c - c_v$  LHPs. If  $c = c_v$  there is no such triple in  $P(x', y_1)$ .
  - (b) If a triple of the form  $(x' x_1, b_1 y_1)$  and weight  $wt'$  is present as an HT in  $P^*(x', y_1)$ , then it represents the exact same number of LHPs  $c - c_v$  of the corresponding triple in  $P(x', y_1)$ . This is exactly the number of HPs of the form  $(x' x_1, b_1 y_1)$  and weight  $wt'$  in  $G - \{v\}$ .
  - (c)  $x' \in L(x_1, b_1 y_1)$ ,  $y_1 \in R(x' x_1, b_1)$  and  $(x' x_1, b_1 y_1) \in \text{Marked-Tuples}$  iff  $c_{\bar{v}} > 0$ . If the triple  $(x' x_1, b_1 y_1)$  is an HT, a similar statement holds for  $x' \in L^*(x_1, y_1)$  iff  $P^*(x_1, y_1) \neq \emptyset$ , and  $y_1 \in R^*(x', b_1)$  iff  $P^*(x', b_1) \neq \emptyset$ .
  - (d) A triple corresponding to  $(x' x_1, b_1 y_1)$  with weight  $wt'$  and counts  $c_v$  is in  $H_c$ . A similar assertion holds for  $y' \in R(x_1 a_1, y_1)$ .
  - (e) The tuple system does not contain any TLHT in  $P(x', y_1)$  and THT in  $P^*(x', y_1)$  of the form  $(x' x_1, b_1 y_1)$  and weight  $wt'$ .
3. For any key  $[wt_2, x_2, y_2] \geq [wt, x, y]$ , let  $G$  contain  $c > 0$  LHPs of weight  $wt_2$  of the form  $(x_2 a_2, b_2 y_2)$ . Further, let  $c_v$  (resp.  $c_{\bar{v}}$ ) denote the number of such LHPs that pass through  $v$  (resp. do not pass through  $v$ ). Here  $c_v + c_{\bar{v}} = c$ . Then the tuple  $(x_2 a_2, b_2 y_2) \in \text{Marked-Tuples}$ , iff  $c_{\bar{v}} > 0$  and a triple for  $(x_2 a_2, b_2 y_2)$  is present in  $H_c$

**Initialization:** We start by showing that the invariants hold before the first loop iteration. The min-key triple in  $H_c$  has key  $[0, v, v]$ . Invariant assertion 1 holds since we inserted into  $H_c$  the trivial triple of weight 0 corresponding to the vertex  $v$  and that is the only triple of such key. Moreover, since we do not represent trivial paths containing the single vertex, no counts need to be decremented. Since we assume positive edge weights, there are no LHPs in  $G$  of weight less than zero. Thus all the points of invariant assertion 2 hold trivially. Invariant assertion 3 holds since  $H_c$  does not contain any triple of weight  $> 0$  and we initialized Marked-Tuples to empty.

**Maintenance:** Assume that the invariants are true before an iteration  $k$  of the loop. We prove that the invariant assertions remains true before the next iteration  $k + 1$ . Let the min-key triple at the beginning of the  $k$ -th iteration be  $[wt_k, x_k, y_k]$ . By invariant assertion 1, we know that for any  $a_i, b_j$ , if there exists a triple  $\gamma$  of the form  $(x_k a_i, b_j y_k)$  of weight  $wt_k$  representing *count* paths going through  $v$ , then it is present in  $H_c$ . Now consider the set of triples with key  $[wt_k, x_k, y_k]$  which we extract in the set  $S$  (Step 4, Algorithm 3). We consider left-extensions of triples in  $S$ ; symmetric arguments apply for right-extensions. Consider for a particular  $b$ , the set  $S_b \subseteq S$  of triples of the form  $(x_k \times, b y_k)$  and let  $fcount'$  denote the sum of the counts of the paths represented by triples in  $S_b$ . Let  $x' \in L(x_k, b y_k)$  be a left extension; our goal is to generate the triple  $(x' x_k, b y_k)$  with count  $fcount'$  and weight  $wt' = wt_k + \mathbf{w}(x', x_k)$ . However, we generate such triple only if it has not been generated by a right-extension of another set of paths. We observe that the paths of the form  $(x' x_k, b y_k)$  can be generated by right extending to  $y_k$  the set of triples of the form  $(x' x_k, \times b)$ . Without loss of generality assume that the triples of the form  $(x' x_k, \times b)$  have a key which is greater than the key  $[wt_k, x_k, y_k]$ . Thus, at the beginning of the  $k$ -th iteration, by invariant assertion 3, we know that  $(x' x_k, b y_k) \notin \text{Marked-Tuples}$ . Steps 9–12, Algorithm 3 create a triple of the form  $(x' x_k, b y_k)$  of weight  $wt'$ . The generated triple can be an LST or a TLHT in  $P$ . In the first case the condition at step 11, Algorithm 3 holds and we remove  $\gamma'$  by decrementing  $fcount'$  many paths from the appropriate triple in  $P(x', y_k)$ . Moreover, if the condition does not hold, we just encountered a TLHT: in this case we completely remove this TLHT from  $P$  in step 18, Algorithm 3. Moreover, if the generated triple is also contained in  $P^*(x', y_k)$ , we check if it is an ST or a THT using step 23, Algorithm 3. In the case of an ST we normally decrement  $fcount'$  paths from the appropriate triple in  $P^*(x', y_k)$ , otherwise we completely remove the THT from  $P^*(x', y_k)$  in

step 26, Algorithm 3. This establishes invariant assertions 2a, 2b and 2e. In addition, if there are no LSPs in  $G$  of the form  $(x'x_k, by_k)$  which do not pass through  $v$ , we delete  $x'$  from  $L(x_k, by_k)$  and delete  $y_k$  from  $R(x'x_k, b)$  (Step 16, Algorithm 3). On the other hand, if there exist LSPs in  $G$  of the form  $(x'x_k, by_k)$ , then  $x'$  (resp.  $y_k$ ) continues to exist in  $L(x_k, by_k)$  (resp. in  $R(x'x_k, b)$ ). Further, we add the tuple  $(x'x_k, by_k)$  to Marked-Tuples and observe that the corresponding triple is already present in  $H_c$  (Step 14, Algorithm 3). Similarly, if the generated triple  $(x'x_k, by_k)$  is an ST, then we check if  $P^*(x_k, y_k) = \emptyset$  and  $P^*(x', b) = \emptyset$  in order to delete  $x'$  from  $L^*(x_k, y_k)$  and  $y_k$  from  $R^*(x', b)$ . By the loop invariant, invariant assertions 2c and 2d were true for every key  $< [wt_k, x_k, y_k]$  and by the above steps we ensure that these invariant assertions hold for every key  $= [wt_k, x_k, y_k]$ . Thus, invariant assertions 2c and 2d are true at the beginning of the  $(k+1)$ -th iteration. Note that any triple that is generated by a left extension (or symmetrically right extension) is inserted into  $H_c$  as well as into Marked-Tuples. This establishes invariant assertion 3 at the beginning of the  $(k+1)$ -th iteration.

Finally, to see that invariant assertion 1 holds at the beginning of the  $(k+1)$ -th iteration, let the min-key at the  $(k+1)$ -th iteration be  $[wt_{k+1}, x_{k+1}, y_{k+1}]$ . Observe that triples with weight  $wt_{k+1}$  starting with  $x_{k+1}$  and ending in  $y_{k+1}$  can be created either by left extending or right extending the triples of smaller weight. And since for each of iteration  $\leq k$ , invariant assertion 2 holds for any extension, we conclude that invariant assertion 1 holds at the beginning of the  $(k+1)$ -th iteration. This finishes our maintenance step.

**Termination:** The condition to exit the loop is  $H_c = \emptyset$ . Because invariant assertion 1 maintains in  $H_c$  all the triples already processed, then  $H_c = \emptyset$  implies that there are no other triples to extend in the graph  $G$  that contain the updated node  $v$ . Moreover, because of invariant assertion 1, every triple containing the node  $v$  inserted into  $H_c = \emptyset$ , has been correctly decremented from the tuple system. Also, since invariant assertion 2e holds, every THT and TLHT containing the node  $v$  has been completely deleted from the tuple system. Finally, for invariant assertion 2c, the sets  $L, L^*, R, R^*$  are correctly maintained. This completes the proof.  $\square$

For FULLY-FIXUP, we first show that Algorithm 4 computes the correct distances for all the SPs in the updated graph  $G'$  (Lemma 16). Moreover, we process all the *new* SPs in  $G'$  (Lemma 18). Finally, we show that data structures and counts are correctly maintained after the algorithm (Lemma 20). Here we use the notion of a *fresh* LHT for a triple that represents at least one path that is in  $P$  but not in  $P^*$ . We will consider fresh triples in Lemma 18 and Observation 19.

**Invariant 15.** *During the execution of Algorithm 4, for any pair  $(x, y)$ , consider the first extraction from  $H_f$  of a set of triples from  $x$  to  $y$ , and let their weight be  $wt$ . Then  $wt$  is the shortest path distance from  $x$  to  $y$  in the updated graph  $G'$ .*

**Lemma 16.** *Algorithm 4 maintains Invariant 15.*

*Proof.* Suppose for a contradiction that the invariant is violated at some extraction. Consider the earliest event in which the first set of triples  $S'$  of weight  $\hat{wt}$ , extracted for some pair  $(x, y)$ , does not contain STs in  $G'$ . Let  $\gamma = ((xa, by), wt, count)$  be a triple in  $G'$  that represents at least one shortest path from  $x$  to  $y$  in  $G'$ , with  $wt < \hat{wt}$ . The triple cannot be in  $P(x, y)$  at the beginning of fixup otherwise it (or another triple with same weight  $wt$ ) would have been inserted in  $H_f$  during step 5 of Algorithm 4. Moreover,  $\gamma$  cannot be in  $H_f$  otherwise it would have been extracted before any triple of weight  $\hat{wt}$  in  $S'$ ; hence  $\gamma$  must be a *new* LST generated by the algorithm. Since all the edges incident to  $v$  are added to  $H_f$  during step 3 of Algorithm 4, then  $\gamma$  must represent SPs of at least two edges. We define  $left(\gamma)$  as the set of LSTs of the form  $((xa, c_i b), wt - \mathbf{w}(b, y), count_{c_i})$  that represent all the LSPs in the left tuple  $((xa, b), wt - \mathbf{w}(b, y))$ ; similarly we define  $right(\gamma)$  as the set of LSTs of the form  $((adj, by), wt - \mathbf{w}(x, a), count_{a_j})$  that represent all the LSPs in the right tuple  $((a, by), wt - \mathbf{w}(x, a))$ .

Observe that since  $\gamma$  is an ST, all the LSTs in  $left(\gamma)$  and  $right(\gamma)$  are also STs. A triple in  $left(\gamma)$  and a triple in  $right(\gamma)$  cannot be present in  $P^*$  together at the beginning of fixup. In fact, if at least one triple from both sets is present in  $P^*$  at the beginning of fixup, then the last one inserted during the fixup triggered by the previous update, would have generated an LST of the form  $((xa, by), wt)$  automatically inserted and



thus present in  $P$  at the beginning of fixup (a contradiction). Thus either there is no triple in  $left(\gamma)$  in  $P^*$ , or there is no triple in  $right(\gamma)$  in  $P^*$ .

Assume w.l.o.g. that no triple in  $right(\gamma)$  is in  $P^*$ . Since edge weights are positive,  $wt - \mathbf{w}(x, a) < wt < \hat{wt}$ , and because all the extractions before  $\gamma$  were correct, then the triples in  $right(\gamma)$  were correctly extracted from  $H_f$  and placed in  $P^*$  before the wrong extractions in  $S'$ . If at least one triple in  $left(\gamma)$  is in  $P^*$  then the fixup would generate the tuple  $((xa, by), wt)$  and place it in  $P$  and  $H_f$  (Steps 24–34, Algorithm 4). Otherwise, since  $wt - \mathbf{w}(b, y) < wt < \hat{wt}$ , the triples in  $left(\gamma)$  were discovered by the algorithm before the wrong extractions in  $S'$ . Moreover the algorithm would generate the tuple  $((xa, by), wt)$  (as right extensions) and place it in  $P$  and  $H_f$  (because at least one triple in  $right(\gamma)$  is already in  $P^*$ ). Thus, in both cases, a tuple  $((xa, by), wt)$  should have been extracted from  $H_f$  before any triple in  $S'$ . A contradiction.  $\square$

**Invariant 17.** *The set  $S$  of triples constructed in Steps 9–23 of Algorithm 4 represents all the new shortest paths from  $x$  to  $y$ .*

**Lemma 18.** *Algorithm 4 maintains Invariant 17.*

*Proof.* Any new SP from  $x$  to  $y$  is of the following three types:

1. a single edge containing the vertex  $v$  (such a path is added to  $P(x, y)$  and  $H_f$  in Step 3)
2. a path generated via left/right extension of some shortest path previously extracted from  $H_f$  during the execution of Algorithm 4 (this generated path is added to  $P(x, y)$  and  $H_f$  in Step 29 and an analogous step in right-extend).
3. a path that was an LSP but not an SP before the update and is an SP after the update.

In type (1) and (2) above any new SP from  $x$  to  $y$  which is added to  $H_f$  is also added to  $P^*(x, y)$ . However, amongst the several triples representing paths of the type (3) listed above, only one candidate triple will be present in  $H_f$ . Thus we conclude that, for a given  $x, y$ , when we extract from  $H_f$  a type (3) triple of weight  $wt$ ,  $P(x, y)$  could contain a superset of triples with the same weight  $wt$  that are not present in  $H_f$ . We now consider the two cases the algorithm deals with.

- $P^*(x, y)$  increased its min-weight, when the first set of triples for  $x, y$  is extracted from  $H_f$ . This is the only case where we could restore historical triples, or process fresh triples from scratch because they are not yet in  $P^*$  or they are present in  $P^*$  with a lower count than the corresponding triple in  $P$  (note that this condition is triggered only by decremental updates). To do that we process all the min-weight triples in  $P(x, y)$ , but before we really add a triple in  $S$  for further extensions, we check if it is present in  $P^*$  with a lower count (Step 16, Algorithm 4), or it is not present in  $P^*$  (Step 13, Algorithm 4). By the above argument, we consider all the new STs from  $x$  to  $y$  present in  $P(x, y)$ . Therefore it suffices to argue that all of them contains *new* shortest paths to be processed. Suppose for contradiction that some triple  $\gamma$  does not contains *new* shortest paths. Thus,  $\gamma$  was a ST before the update and already in  $P^*$  with at least one path not going through  $v$ . However, since cleanup only removes paths that contain  $v$ , the triple  $\gamma$  remained in  $P^*(x, y)$  after the FULLY-CLEANUP phase. This contradicts the fact that  $P^*(x, y)$  increased its min-weight.
- $P^*(x, y)$  didn't change its min-weight when the first set of triples for  $x, y$  is extracted from  $H_f$ . Let the weight of triples in  $P^*(x, y)$  be  $wt$ . This implies that the shortest path distance from  $x$  to  $y$  before and after the update is  $wt$ . Both in the case of incremental and decremental updates, all the new paths that we need to consider from  $x$  to  $y$  are going through the updated node  $v$ . By construction of the Algorithm 4, every triple containing the updated node  $v$  is always placed into  $H_f$ . Thus it suffices to consider only triples in  $H_f$ .

$\square$

**Observation 19.** *During the execution of Algorithm 4, consider a THT  $\tau$  that becomes shortest. If  $\tau$ 's corresponding triple in  $P$  is not fresh, then it is simply restored (not processed); otherwise  $\tau$ 's count is replaced with the updated count from  $P$  and it is extended anew.*

*Proof.* When we restore an existing HT  $\tau$  from  $P^*$ , we always check if its corresponding triple in  $P$  contains more paths (Step 16, Algorithm 4) or the counts match. In the first case  $\tau$  in  $P^*$  is carrying an obsolete number of SPs and is therefore replaced with the correct count in  $P$  and extended anew (Step 17, Algorithm 4). Otherwise it is still representing the correct number of SPs to be restored and it is not processed.  $\square$

**Lemma 20.** *After the execution of Algorithm 4 (FULLY-FIXUP), for any  $(x, y) \in V$ , the STs in  $P^*(x, y)$  (LSTs in  $P(x, y)$ ) represent all the SPs (LSPs) from  $x$  to  $y$  in the updated graph. Also, the sets  $L, L^*, R, R^*$  are correctly maintained.*

*Proof.* We prove the lemma statement by showing the following loop invariant.

**Loop Invariant:** At the start of each iteration of the while loop in Step 6 of Algorithm 4, assume that the first triple in  $H_f$  to be extracted and processed has min-key =  $[wt, x, y]$ . Then the following properties hold about the tuple system and  $H_f$ .

1. For any  $a, b \in V$ , if  $G'$  contains  $c_{ab}$  SPs of form  $(xa, by)$  and weight  $wt$ , then  $H_f$  contains a triple of form  $(xa, by)$  and weight  $wt$  to be extracted and processed. Further, a triple  $\gamma = ((xa, by), wt, c_{ab})$  is present in  $P(x, y)$ .
2. Let  $[\hat{wt}, \hat{x}, \hat{y}]$  be the last key extracted from  $H_f$  and processed before  $[wt, x, y]$ . For any key  $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$ , let  $G'$  contain  $c > 0$  number of LHPs of weight  $wt_1$  of the form  $(x_1a_1, b_1y_1)$ . Further, let  $c_{new}$  (resp.  $c_{old}$ ) denote the number of these LHPs that are *new* (resp. not *new*). Here  $c_{new} + c_{old} = c$ . If  $c_{new} > 0$  then,
  - (a) there is an LHT in  $P(x_1, y_1)$  of the form  $(x_1a_1, b_1y_1)$  and weight  $wt_1$  that represents  $c$  LHPs.
  - (b) If a triple of the form  $(x_1a_1, b_1y_1)$  and weight  $wt_1$  is present as an HT in  $P^*$ , then it represents the exact same count of  $c$  HPs of its corresponding triple in  $P$ . This is exactly the number of HPs of the form  $(x_1a_1, b_1y_1)$  and weight  $wt_1$  in  $G'$ .
  - (c)  $x_1 \in L(a_1, b_1y_1)$ ,  $y_1 \in R(x_1a_1, b_1)$ , and if the triple of the form  $(x_1a_1, b_1y_1)$  and weight  $wt_1$  is also shortest then  $x_1 \in L^*(a_1, y_1)$ ,  $y_1 \in R^*(x_1, b_1)$ . Further,  $(x_1a_1, b_1y_1) \in \text{Marked-Tuples}$  iff  $c_{old} > 0$ .
  - (d) If  $c_{new} > 0$ , for every  $x' \in L(x_1, b_1y_1)$ , an LHT corresponding to  $(x'x_1, b_1y_1)$  with weight  $wt' = wt_1 + \mathbf{w}(x', x_1) \geq wt$  and counts equal to the sum of new paths represented by its constituents, is in  $H_f$  and  $P$ . A similar assertion holds for  $y' \in R(x_1a_1, y_1)$ .
3. For any key  $[wt_2, x_2, y_2] \geq [wt, x, y]$ , let  $G'$  contain  $c > 0$  number of LHPs of weight  $wt_2$  of the form  $(x_2a_2, b_2y_2)$ . Further, let  $c_{new}$  (resp.  $c_{old}$ ) denote the number of such LHPs that are *new* (resp. not *new*). Here  $c_{new} + c_{old} = c$ . Then the tuple  $(x_2a_2, b_2y_2) \in \text{Marked-Tuples}$ , iff  $c_{old} > 0$  and  $c_{new}$  paths have been added to  $H_f$  by some earlier iteration of the while loop.

Initialization and Maintenance for the 3 invariant assertions are similar to the proof of Lemma 14.

**Termination:** The condition to exit the loop is  $H_f = \emptyset$ . Because invariant assertion 1 maintains in  $H_f$  the first triple to be extracted and processed, then  $H_f = \emptyset$  implies that there are no triples, formed by a valid left or right extensions, that contain *new* SPs or LSPs, that need to be added or restored in the graph  $G$ . Moreover, because of invariant assertions 2a and 2b, every triple containing the node  $v$ , extracted and processed before  $H_f = \emptyset$ , has been added or restored with its correct count in the tuple system. Finally, for invariant assertion 2c, the sets  $L, L^*, R, R^*$  are correctly maintained. This completes the proof of the loop invariant.

By Lemma 18, all the new SPs in  $G'$  are placed in  $H_f$  and processed by the algorithm and hence are in  $P^*$  after the execution of Algorithm 4. Moreover, for a pair  $(x, y)$ , the check in Step 9 of Algorithm 4 would fail if the distance from  $x$  to  $y$  doesn't change after the update. Thus the old SPs from  $x$  to  $y$  will remain in  $P^*(x, y)$ . Hence, after Algorithm 4 is executed, every SP in  $G'$  is in its corresponding  $P^*$ .

Since every LST of the form  $(xa, by)$  in  $G'$  is formed by a left extension of a set of STs of the form  $(a*, by)$  (Steps 24–34, Algorithm 4), or a right extension of a set of the form  $(xa, *b)$  (analogous steps for right extensions), and all the STs are correctly maintained by the algorithm, then all the LSTs are correctly maintained at the end of the fixup algorithm. This completes the proof of the Lemma.  $\square$

## B Proofs of Lemmas 6 and 7

**Lemma 6.** *Consider a sequence of  $r$  calls to FULLY-UPDATE on a graph with  $n$  vertices. Let  $C$  be the maximum number of tuples in the tuple system that can contain a path through a given vertex, and let  $D$  be the maximum number of tuples that can be in the tuple system at any time. Then FULLY-UPDATE executes the  $r$  updates in  $O((r \cdot (n^2 + C) + D) \cdot \log n)$  time.*

*Proof.* We bound the cost of FULLY-UPDATE by classifying each triple  $\gamma$  as one of the following disjoint types:

- **Type-0 (contains-v):**  $\gamma$  represents at least one path containing vertex  $v$ .
- **Type-1 (new-LHT):**  $\gamma$  was not an LHT before the update but is an LHT after the update, and no path in  $\gamma$  contains  $v$ .
- **Type-2 (new-HT-old-LHT):**  $\gamma$  is an HT after the update, and  $\gamma$  was an LHT but not an HT before the update, and no path in  $\gamma$  contains  $v$ .
- **Type-3 (renew-ST):**  $\gamma$  was a THT before the update and it is restored as a ST after the update, and no path in  $\gamma$  contains  $v$ .
- **Type-4 (new-LHT-old-LHT):**  $\gamma$  was an LHT before the update and continues to be an LHT after the update, and no path in  $\gamma$  contains  $v$ .

The number of Type-0 triples, processed in Steps 20–23, Algorithm 4, is at most  $C$ . The number of Type-1 triples, processed in Steps 24–34, Algorithm 4, is addressed by amortizing over the entire update sequence as described in the paragraph below. For Type-2 triples, processed in Steps 9–18, Algorithm 4, we observe that after such a triple becomes an HT, it is not removed from  $P^*$  unless a real or dummy update is triggered on a vertex that lies in it. But in such an update this would be counted as a Type-0 triple. Further, each such Type-2 triple is examined only a constant number of times (in Steps 9–18, Algorithm 4), because after they are inserted into  $P^*$  the bit  $\beta$  associated changes to 1 and they will not be processed again by fixup, unless the number of paths they represent is changed. Hence we charge each access to a Type-2 triple to the step in which it was created as a Type-1 triple. For Type-3 triples, also processed in Steps 9–18, Algorithm 4, we distinguish two cases: if  $\gamma$  didn't change its count in  $P$  after it became a THT then its flag is  $\beta = 1$  and it is present in  $P^*$  with the correct count. Thus the fixup algorithm will not process it. If  $\gamma$  changed its count in  $P$  while it was a THT then its flag is  $\beta = 0$  and we can charge the processing of  $\gamma$  (if extracted from  $H_f$ ) to the sub-triple  $\gamma'$  generated from the updated node that increased the count of  $\gamma$ ; in other words there was an ST  $\gamma'$ , created in a previous update, whose extensions added to the count of  $\gamma$ . Observe that, triples in  $P^*$  that are not placed initially in  $H_f$  and have  $\beta = 1$  in  $P$  (no additional path was added to that triple) are not examined in any step of fixup, so no additional Type-3 triples are examined. For Type-4, we note that for any  $x, y$  we add exactly one candidate min-key triple from  $P(x, y)$  to  $H_f$  (in Step 5, Algorithm 4), hence initially there are at most  $n^2$  such triples in  $H_f$ , any of which could be Type-4. Moreover, we never process an old LHT which is not a new HT so no additional Type-4 triples are examined during fixup. Thus the number of triples examined by a call to fixup is  $C$  plus  $X$ , where  $X$  is the number of new triples fixup adds to the tuple system. (This includes an  $O(1)$  credit placed on each new LHT for a possible later conversion to an HT.)

Let  $r$  be the number of updates in the update sequence. Since triples are removed only in cleanup, at most  $O(r \cdot C)$  triples are removed by the cleanups. There can be at most  $D$  triples remaining at the end of the sequence, hence the total number of new triples added by all fixups in the update sequence is  $O(r \cdot C + D)$ . Since the time taken to access a triple is  $O(\log n)$  due to the data structure operations, and we examine at least  $n^2$  triples at each round, the total time spent by fixup over  $r$  updates is  $O((r \cdot (n^2 + C) + D) \cdot \log n)$ .  $\square$

**Lemma 7.** *At each step  $t$ , the tuple system for Algorithm FULLY-UPDATE maintains a subset of HTs and LHTs that includes all STs and LSTs for step  $t$ , and further, for every LHT triple  $((x, a), (y, b), wt, count)$  in step  $t$ , there are HTs  $(a, y)$  and  $(x, b)$  with weights  $wt - w(x, a)$  and  $wt - w(b, y)$  respectively, in that step.*

*Proof.* For the first part see Appendix A.1 (Lemmas 14 and 20). For the second part, if a triple  $\gamma = ((xa, by), wt, count)$  is present in step  $t$ , then  $\gamma$  was generated during FULLY-FIXUP of some step  $t' \leq t$ . By the construction of our algorithm, at the end of step  $t'$ , the tuple system contains at least one HT of the form  $(a*, by)$  and one of the form  $(xa, *b)$  with weights  $wt - w(x, a)$  and  $wt - w(b, y)$  respectively. W.l.o.g. suppose that the set  $S$  of all the HTs of the form  $(a*, by)$  and weight  $wt - w(x, a)$  are removed during FULLY-CLEANUP at some step  $t'' \leq t$ , then since these HTs are the right constituents of  $\gamma$ , when  $S$  is left extended to  $x$  in FULLY-CLEANUP then exactly  $count$  paths will be removed from  $\gamma$  making the triple disappear from the tuple system. Thus at step  $t$ , at least one HT of the form  $(a*, by)$  with weight  $wt - w(x, a)$  must be in  $P^*$ . Similarly, there must be an HT of the form  $(xa, *b)$  with weight  $wt - w(b, y)$  in  $P^*$  at step  $t$ .  $\square$